CSE574 Introduction to Machine Learning

Basic Practice

Jue Guo

University at Buffalo

February 1, 2024

Outline

Until now, I only mentioned in passing some issues that a data analyst needs to consider when working on a machine learning problem: feature engineering, overfitting, and hyperparameter tuning.

- For this session, we talk about these and other challenges that have to be addressed before you can type `model = LogisticRegression().fit(x,y)` in scikit-learn.

Feature Engineering

When a product manager tells you "We need to be able to predict whether a particular customer will stay with us. Here are the logs of customers' interactions with our product for five years." you cannot just grab the data, load it into a library and get a prediction. **You need to build a dataset first.**

Remember from the first class that the dataset is the collection of **labeled examples** $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$. Each element $\mathbf{x}_i$ among N is called a **feature vector**. A feature vector is a vector in which each dimension $j = 1, \ldots, D$ contains a value that describes the example somehow. That value is called a **feature** and is denoted as $x^{(j)}$.

The problem of transforming raw data into a dataset is called **feature engineering**. For most practical problems, feature engineering is a labor-intensive process that demands from the data analyst a lot of creativity and, preferably, domain knowledge.

For example, to transform the logs of user interaction with a computer system, one could create features that contain information about the user and various statistics extracted from the logs.

- For each user, one feature would contain the price of the subscription; other features would contain the frequency of connections per day, week and year.
- Another feature would contain the average session duration in seconds or the average response time for one request, and so on. Everything measurable can be used as a feature.

The role of the data analyst is to create *informative* features: those would allow the learning algorithm to build a model that does a good job of predicting labels of the data used for training. Highly informative features are also called features with high *predictive power*.

- For example, the average duration of a user's session has high predictive power for the problem of predicting whether the user will keep using the application in the future.

We say that a model has a **low bias** when it predicts the training data well. That is, the model makes few mistakes when we use it to predict labels of the examples used to build the model.

One-Hot Encoding

Some learning algorithms only work with numerical feature vectors. When some feature in your dataset is categorical, like “colors" or “days of the week," you can transform such a categorical feature into several binary ones.

If your example has a categorical feature "colors" and this feature has three possible values: "red," "yellow," "green," you can transform this feature into a vector of three numerical values:

$$\begin{aligned} \text{red} &= [1, 0, 0] \\ \text{yellow} &= [0, 1, 0] \\ \text{green} &= [0, 0, 1] \end{aligned}$$

By doing so, you increase the dimensionality of your feature vectors. You should not transform red into 1 , yellow into 2 , and green into 3 to avoid increasing the dimensionality because that would imply that there’s an order among the values in this category and this specific order is important for the decision making. If the order of a feature’s values is not important, using ordered numbers as values is likely to confuse the learning algorithm, because the algorithm will try to find a regularity where there’s no one, which may potentially lead to overfitting.

Binning

An opposite situation, occurring less frequently in practice, is when you have a numerical feature but you want to convert it into a categorical one.

- **Binning** (also called **bucketing**) is the process of converting a continuous feature into multiple binary features called bins or buckets, typically based on value range.

For example, instead of representing age as a single real-valued feature, the analyst could chop ranges of age into discrete bins: all ages between 0 and 5 years-old could be put into one bin, 6 to 10 years-old could be in the second bin, 11 to 15 years-old could be in the third bin, and so on.

Let feature $j = 4$ represent age.

- By applying binning, we replace this feature with the corresponding bins. Let the three new bins, "age_bin1", "age_bin2" and "age_bin3" be added with indexes $j = 123$, $j = 124$ and $j = 125$ respectively (by default the values of these three new features are 0). Now if $x_i^{(4)} = 7$ for some example $\mathbf{x}_i$, then we set feature $x_i^{(124)}$ to 1 ; if $x_i^{(4)} = 13$, then we set feature $x_i^{(125)}$ to 1 , and so on.

In some cases, a carefully designed binning can help the learning algorithm to learn using fewer examples. It happens because we give a "hint" to the learning algorithm that if the value of a feature falls within a specific range, the exact value of the feature doesn't matter.

Normalization

Normalization is the process of converting an actual range of values which a numerical feature can take, into a standard range of values, typically in the interval $[-1, 1]$ or $[0, 1]$.

For example, suppose the natural range of a particular feature is 350 to 1450 . By subtracting 350 from every value of the feature, and dividing the result by 1100 , one can normalize those values into the range $[0, 1]$.

More generally, the normalization formula looks like this:

$$\bar{x}^{(j)} = \frac{x^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}}$$

where $\min^{(j)}$ and $\max^{(j)}$ are, respectively, the minimum and the maximum value of the feature j in the dataset.

Why do we normalize? Normalizing the data is not a strict requirement. However, in practice, it can lead to an increased speed of learning. Remember the gradient descent example.

- Imagine you have a two-dimensional feature vector. When you update the parameters of $w^{(1)}$ and $w^{(2)}$, you use partial derivatives of the mean squared error with respect to $w^{(1)}$ and $w^{(2)}$. If $x^{(1)}$ is in the range $[0, 1000]$ and $x^{(2)}$ the range $[0, 0.0001]$, then the derivative with respect to a larger feature will dominate the update.

Additionally, it's useful to ensure that our inputs are roughly in the same relatively small range to avoid problems which computers have when working with very small or very big numbers (known as **numerical overflow**).

Standardization

Standardization (or **z-score normalization**) is the procedure during which the feature values are rescaled so that they have the properties of a *standard normal distribution* with $\mu = 0$ and $\sigma = 1$, where μ is the mean (the average value of the feature, averaged over all examples in the dataset) and σ is the standard deviation from the mean.

Standard scores (or z-scores) of features are calculated as follows:

$$\hat{x}^{(j)} = \frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}}$$

You may ask when you should use normalization and when standardization. There's no definitive answer to this question. Usually, if your dataset is not too big and you have time, you can try both and see which one performs better for your task.

If you don't have time to run multiple experiments, as a rule of thumb:

- unsupervised learning algorithm, in practice, more often benfit from standardization from normalization;
- standardization is also preferred for a feature if the values this feature takes are distributed close to a normal distribution (so-called bell curve);
- again, standardization is preferred for a feature if it can sometimes have extremely high or low values (outliers); this is because normalization will "squeeze" the normal values into a very small range;
- in all other cases, normalization is preferable.

Feature rescaling is usually beneficial to most learning algorithms. However, modern implementations of the learning algorithms, which you can find in popular libraries, are robust to features lying in different ranges.

Dealing with Missing Features

In some cases, the data comes to the analyst in the form of a dataset with features already defined. In some examples, values of some features can be missing. That often happens when the dataset was handcrafted, and the person working on it forgot to fill some values or didn't get them measured at all.

The typical approaches of dealing with missing values for a feature include:

- removing the examples with missing features from the dataset (that can be done if your dataset is big enough so you can sacrifice some training examples);
- using a learning algorithm that can deal with missing feature values (depends on the library and a specific implementation of the algorithm);
- using a **data imputation** technique.

Data Imputation Techniques

One data imputation technique consists in replacing the missing value of a feature by an average value of this feature in the dataset:

$$\hat{x}^{(j)} \leftarrow \frac{1}{N} \sum_{i=1}^{N} x_i^{(j)}$$

Another technique is to replace the missing value with a value outside the normal range of values. For example, if the normal range is $[0, 1]$, then you can set the missing value to 2 or -1 . The idea is that the learning algorithm will learn what is best to do when the feature has a value significantly different from regular values. Alternatively, you can replace the missing value by a value in the middle of the range.

- For example, if the range for a feature is $[-1, 1]$, you can set the missing value to be equal to 0 . Here, the idea is that the value in the middle of the range will not significantly affect the prediction.

A more advanced technique is to use the missing value as the target variable for a regression problem.

- You can use all remaining features $\left[x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(j-1)}, x_i^{(j+1)}, \ldots, x_i^{(D)}\right]$ to form a feature vector $\hat{\mathbf{x}}_i$, set $\hat{y}_i \leftarrow x^{(j)}$, where j is the feature with a missing value.

Then you build a regression model to predict $\hat{y}$ from $\hat{\mathbf{x}}$. Of course, to build training examples ($\hat{\mathbf{x}}$, $\hat{y}$), you only use those examples from the original dataset, in which the value of feature j is present.

Finally, if you have a significantly large dataset and just a few features with missing values, you can increase the dimensionality of your feature vectors by adding a binary indicator feature for each feature with missing values. Let's say feature $j = 12$ in your D-dimensional dataset has missing values. For each feature vector $\mathbf{x}$, you then add the feature $j = D + 1$ which is equal to 1 if the value of feature 12 is present in $\mathbf{x}$ and 0 otherwise. The missing feature value then can be replaced by 0 or any number of your choice.

Questions?